The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Working Paper 256 June, 1984

# CHAPTER AND VERSE PROGRAM DESCRIPTION

Elizabeth K. Turrisi

## ABSTRACT

The design of a program is rarely a straightforward mapping from the problem solution to the code. More frequently, fragments of high level concepts are distributed over one or more modules such that it is hard to identify the fragments which belong to one particular concept. These mappings have to be untangled and described in order to give a complete picture of how the program implements the ideas.

The Chapter and Verse method of program description emphasizes the high level concepts which underlie a program, and the relationship between these concepts and the low level structure of program code. The organization of the description is similar to that of a textbook. The Chapter and Verse description aids in the use, modification, and evaluation of computer programs by promoting a full understanding of the programs.

# ACKNOWLEDGEMENTS

I would like to thank Chuck Rich for encouraging my interest in Computer Science and my thesis subject, and my parents and Linda Linda Zelinka for proof-reading and support. My special thanks go to my thesis supervisor, Dick Waters, for all the ideas and suggestions which he contributed to my thesis.

# 1. INTRODUCTION

A thorough program description can be of great benefit in the design, development, modification, use, and evaluation of a computer program. This paper explores a method of program description, called "Chapter and Verse".

The design of a program is rarely a straightforward mapping from the problem solution to the code. More frequently, fragments of high level concepts are distributed over one or more modules such that it is hard to identify the fragments as belonging to one particular concept. (Moreover, it is quite difficult to invert the process and discover the high level concepts from the code.) These mappings have to be untangled and described in order to give a complete picture of how the program implements the ideas.

The high level ideas and the algorithms chosen to implement the ideas make up the *conceptual structure.* The conceptual structure is the way the problem solution looks before it is transformed into computer code. The mapping of the conceptual structure to the code is usually web-like rather than linear. Since the conceptual structure is the basis of the entire program, it is essential that it is explained well.

The description must first present a fairly comprehensive view of the problem domain to give the reader enough background knowledge so the high level ideas are clear. It must be flexible enough to handle the web-like relationships between high level ideas and code. The general discussion of the problem should include information about what the problem is and how it was solved.

These properties indicate that the description should be non-linear, include general information about the problem, and have some way to link the general information to the implementation. It has been suggested that these objectives can be met in large part with an organization which follows that of a textbook.

The textbook-like description of computer programs was originally conceived by Kenneth Wilson, a physicist at Cornell University. He initiated the GIBBS Project with the objective of developing a programming environment which will support the organized presentation of a computer program. With such a system, programs will be developed by transformation from the original high level specifications. Since Wilson is a physicist, he is concerned with the difficulties in understanding and modifying a heavily optimized program. Wilson suggests that much of the difficulty is due to the fact that the implementation of any one high level idea is scattered throughout the program. Therefore, the GIBBS Project proposes transformations which will make clear the links between the high level concepts and their implementations.

This paper focuses on an example of the textbook-style program description, Chapter and Verse. A sample description of the text formatter TFORMAT[1] was prepared as a supplement to this paper. The example description may be found in Appendix 1; the TFORMAT program is Appendix 2. At least a brief look at the sample prior to reading this paper is advisable since examples will often refer to that description. (A second reading of the description after reading the paper would be ideal.) The TFORMAT description will be discussed in detail below.

---

1. Liskov, B., CLU Reference Manual, Springer-Verlag, New York, 1981.

# 2. WHY IS GOOD DESCRIPTION OF PROGRAMS IMPORTANT?

There are many reasons why a program should be thoroughly described. A user should understand the purpose of the program and how to take full advantage of the program's capabilities. A modifier of the program has to be aware of the program's background and of the consequences of any modifications. Another reason for understanding a program is that the program can then be evaluated intelligently to see if it meets its requirements and specifications.

## 2.1 Understanding

A description which includes information about the high level concepts behind a computer program can be instrumental in helping someone to understand a program. The reader gains insight into the program by seeing not only what the program does, but why it does what it does. The description includes information about how and why particular algorithms were chosen, and about the options which were not chosen.

A complete understanding of a program makes a user aware of the program's capabilities and limitations. The user gains familiarity with the program and its operation so that he may make better use of the program in doing his work. He is able to determine if the program is appropriate for his particular application.

A user may gain as much background information as he needs in order to use the program to the best benefit. With a thorough understanding of the purpose and operation of the program and the features available, the user has all the information necessary for effective use of the program. He has available to him information which includes descriptions of the purpose of the program, high level discussions of the underlying conceptual structure of the program, idiosyncrasies of the algorithms used and their implementations, motivations for particular implementations, and other points of interest.

## 2.2 Modification

If a program is to be modified, then the program has to be completely understood so that the consequences of any modifications are well understood. If the effects of changing a program are not examined, then the program could end up being inconsistent in code or in concept. The program might also be changed in such a way as to cause it to produce incorrect results as the result of erroneous code or distorted conceptual structure. If the code is inconsistent in some way due to a program modification, then the modification was made incorrectly. It is likely that the incorrect modification was made due to a lack of understanding of the basic purpose and meaning of the program.

A potential modifier of a program is at a disadvantage if he was not part of the program's design process since he has no clear idea of what the program is all about. Even if he was involved in the design, he may not be fully aware of the high level issues which must be considered before making a modification. While the code may be well-documented, this documentation is often very localized in the code. For example, even if the modifier knows what computation is performed by lines 200 - 400, he may not have the slightest idea of the relation is of those lines to the program as a the whole. He does not know what would happen to the conceptual framework of the program if lines 200 - 400 did something completely different. Although understanding the computation is important, it is also important to understand how the lines of code fit into an overall algorithm. The computation is probably relatively easy to discover from brief examination of the code (its documentation exists mainly as a convenience), but it is trickier to figure out how the code fits into the conceptual structure.

Unfortunately, the algorithms and conceptual framework may not be obvious at all. It may not even

be possible to pick out the pieces which fit into a particular thread of meaning without understanding the entire program's code. Naturally, when someone must modify a program he may not feel it is worth the time to study the pages and pages of code in an attempt to discover the underlying concepts. A description of the type explained here fills in the gap. It gives the reader the conceptual framework and the basis for the program, and it shows how the program code itself relates to the basic concepts behind the program.

With this information at hand, anyone can gain understanding of the concepts of the program. This information is useful to the modifier as well as the user. It is also helpful, as discussed below, in evaluating the design to be sure that the design has actually taken account of the needs which the program is to fill.

## 2.3 Evaluation of requirements and specifications fulfillment

One advantage to clearly defining high level issues with a thorough description is that many design issues can be resolved early. This way, design decisions can be made to be consistent with the conceptual framework of the problem solution. Until the problem is fully understood, including the many inevitable subtleties, it is unlikely that a good implementation will be developed. That is to say, if a solution was found, it was found without knowledge of the concepts which the program is implementing and therefore may be in error. This error is not an error in coding, but in design -- the program design might not be the most natural solution of the problem.

Another advantage to an early definition of concepts is that fulfillment of the requirements can be determined. Once the description of the high level ideas is completed, it may be reviewed by the appropriate people to see if the requirements have been correctly interpreted. The description of the relationship between the code and the web of meaning of the program is valuable in assessing how well the program met its requirements, since it becomes clear from the description exactly what the capabilities of the program are, and what functionality can be expected from the program. Additionally, the extendability of the program can be evaluated to be sure that the program is able to grow with the needs of its users.

The high-level chapters of this description would make a valuable addition to a specification document since the information contained in those chapters is necessary for complete understanding of the motivation behind the program. The designer of the program may more easily match the requirements if he understands completely what the conceptual framework of the program is. Understanding of the program allows specifiers, designers, and developers to realize what the important ideas in the program are, and helps them to recognize a natural organization of the program. It also helps them to define the features of the program and to decide how the features can be integrated conceptually with the rest of the program.

# 3. THOROUGH DESCRIPTION OF A COMPUTER PROGRAM

## 3.1 What constitutes the description?

The description is composed of several sections which describe the background and operation of a program. The description includes interesting information about the program, its use, and its history.

In order to understand a program, it is necessary to comprehend, on a high level, what the program is trying to do. Once this is accomplished it is necessary to understand (in a high level sense) how the program will accomplish its task. Then, if low-level knowledge of the program is needed, it is necessary to learn how the actual code of the program implements the execution of the task.

## 3.2 Levels of knowledge

*High level knowledge* is the knowledge about the real world which is relevant to the problem area. In the case of the TFORMAT example, the high level knowledge includes information about text formatters in general, such as how text formatters work, when and how to use them, and what it means to text format a document. Understanding a program on a high level requires a knowledge of the problem which the program is trying to solve. This understanding may extend beyond the actual knowledge reflected in the operation or code of the program.

The *low level knowledge* is the detailed information, usually at the code level. That is, the low level descriptions expose the interesting aspects of the code and the implementations of the algorithms. The low level includes such information as the error handlers in the code, and the sections of code which implement particular parts of algorithms. The low level sections of the description are essential to someone who works with the code, such as a modifier of the program. They are not essential to the users of the program, though the low level sections do provide insight into why the program behaves as it does and how the program may best be put to use on a given problem.

There is also *middle level knowledge*, which is a mixture of the high level and low level. For example, in the TFORMAT example, the middle level sections contain information about commands and error handling information which is not too detailed. The commands are somewhat implementation-specific, so they are considered too detailed for the high level sections. On the other hand, they are sections which are very beneficial to the users of the program so they should not be hidden away with the low level description.

As mentioned earlier, the high level sections help to steer the program designers in the right direction, and provide interesting background information to anyone who is involved with the program. The low level sections link the concepts with the actual program code to show how the concepts have been embodied in the code. Middle level sections do some of each.

## 3.3 Different levels of understanding for different purposes

There are a variety of reasons for wanting to know about a computer program. Some of these reasons require only partial knowledge of the program. Maybe only high level knowledge is required, or maybe only high and middle level knowledge. Perhaps the knowledge required spans from high level to low level, but is only about one aspect of the problem at hand. In these cases, a thorough knowledge is needed in the area of interest, but comprehensive knowledge of every aspect of the program might be overkill.

For example, if someone wants to use a text formatter program, he probably does not want to know about all the internal error handlers. But he may want to know how the justification works. In this case,

he can read the high and middle level descriptions, and selectively read the low level sections which are appropriate. Of course, everyone should read all the high level descriptions, but there are users who do not have to know all the low level details. The TFORMAT description allows them to attain a high level of understanding without wasting time on low level details which do not concern them.

# 4. TRADITIONAL PROGRAM DESCRIPTION

## 4.1 Traditional descriptions are very code-oriented

Traditional descriptions do not pay much attention to the high level concepts which underlie a computer program. As a result, the reader of those descriptions learns of the functions of the code, and the abstractions of the main code modules, but he does not learn why those modules were written the way they were, or how the code of the module corresponds to the real-world model to which the program module belongs.

### 4.1.1 Not enough high level

Program descriptions usually do not provide enough high level information to give the reader adequate background in the problem area. The high level information in a traditional description usually consists of the relationship of the data abstractions to the real world. There is not enough attention paid to the problem area in general, or the significance of key high level factors which are important to understand. The traditional descriptions lack the integration of ideas which could be achieved with a description that provides an overall view of the problem area and the program being described.

### 4.1.2 Not enough linkage from code to high level ideas

The traditional descriptions concentrate on the details of the program implementation or the main abstractions used by the program. Unfortunately, the detailed implementation descriptions do not illustrate how the details correspond to the high level concepts, and the descriptions of the abstractions, by their very nature, do not discuss the details of the implementation. So there is no mechanism for showing how the program code corresponds to the ideas which the program is to implement. This is especially true if the basic algorithms cannot be implemented without crossing the boundaries between the abstractions. In these cases, there is no indication of which sections of code are fulfilling a particular high level purpose.

### 4.1.3 Traditional descriptions describe only sections or modules

A program is usually described according to its *structure*; that is, the *code* is what is described. The manner in which the code is described depends on the structure of the code. It is the computational function of the code which dominates a description, while the high level concepts governing the code are not given much attention.

### 4.1.4 Traditional structure of description is linear

A traditional description of a computer program describes the program structure in a "linear" fashion. That is, the description follows the program in nearly the same order as the code appears physically. The traditional description does not explain the code in an order which reflects the purpose of the code or the high level algorithms. When the descriptions follow the structure of the code they may be talking about one section of code which actually contributes to the implementation of several of

the high level ideas. This is a problem with traditional description of code in which several of the high level ideas or algorithms are mixed together -- there is no clear illustration of the implementation of any one high level idea or algorithm. So, the discussion of the high level concepts is confused if it exists at all.

The TFORMAT description describes the code in the framework of the high level concepts, rather than in the order in which the code appears in a file or the order in which the code executes. However, there *is* some discussion about the structure of the program and why the particular structure evolved as it did, and there is some discussion of the flow of the program.

### 4.1.5 Traditional descriptions are too inflexible in viewpoints

Traditional descriptions are limited in their point of view. All subjects are described from the same point of view, even though one point of view is probably insufficient to clearly explain a complex program. Often material can be much easier to describe and understand from a natural viewpoint, but a traditional description cannot benefit from such a feature since it has no mechanism for including viewpoint changes.

# 5. PURPOSE OF THIS DESCRIPTION

The purpose of the description proposed here is to fill in gaps left by traditional program descriptions. One way in which this description differs from traditional descriptions is that this description discusses the general problem area much more than a traditional description would. Most program descriptions would describe a particular program in detail, with little mention of the general concepts which the program implements. For example, in this description there is much discussion about what a text formatter is, apart from the issues of what the particular text formatter TFORMAT does. The omission of the high level general description of the text formatter world would leave the reader without a feel for the background of the problem and the relevant issues. After reading the first few sections of the description of TFORMAT, the reader will feel confident with the concept of a text formatter and will understand the range of functionality of text formatters and how TFORMAT fits into that range. Striking likenesses and differences between TFORMAT and other text formatters are pointed out to give the reader a feel for the relationship between TFORMAT and other text formatters.

The description may be thought of as holding the answers to the interesting questions which could be asked about the program. There are a broad range of questions which are answered by the program description. Certainly, all questions which are answered by traditional program documentation are answered. The following is a list of important questions:

- What is the purpose of the program?
- How are special terms defined in the context of the program?
- What are the main functions of the program?
- What algorithms were chosen to implement the functions?
- Why were those algorithms used? What were the choices?
- Is there anything peculiar in the way the program works?
- What is the result of using the program?
- How is the program used?
- What type of input does the program require?

The TFORMAT description takes the approach of answering each question from a point of view in which the question can most easily be understood. There is not a strict form for the answer of a question -- the answer depends on the particular question. The section which answers the question "What algorithms were used?" considers the input text to be composed of different "regions", each operated on by a certain set of algorithms. On the other hand, the section which describes the commands sometimes considers the input text to be composed of "lines" or "tokens". The answers given in each case are from different viewpoints, and give the most understandable answer to the question.

This description obviously cannot answer every question which could ever be asked about the program. The description has the objective of promoting understanding and providing a link between the main concepts and the program code -- questions which are not relevant to this goal need not be answered. For example, it is assumed that the readers of the low level chapters will be fluent in the programming language in which the program was written. Therefore it is not necessary for the description to explain every program statement since the reader can easily look at the program. The TFORMAT description does not answer questions such as "What does the twenty-first line of the program compute?". The description does answer such questions as "How is the justification algorithm implemented?".

The information given by the program description is complete enough to give the reader sufficient background and detail to allow him to understand the program and see how it implements the underlying concepts.

# 6. OPTIONS FOR ORGANIZATION OF THE DESCRIPTION

This chapter discusses two ways to organize the information of a program description, a "textbook" approach, and an interactive, on-line approach.

## 6.1 Textbook

### 6.1.1 Explanation of textbook model

It turns out that a textbook has most of the properties which are required. A textbook has the purpose of explaining a subject in full detail such that the student of the text is able to understand the subject well. Textbooks usually have the following properties:

- *The subjects covered in the textbook are discussed on several levels.* Textbooks usually highlight the material first to give the general idea of the subject. After the idea is introduced, the detailed points are covered in the context of the general ideas. This helps to unify the information so that the various topics which are presented can be integrated with the basic concepts.

- *The chapters of a textbook are understandable units.* A chapter may build on some previous material, but it usually has the responsibility for giving the complete presentation of a particular topic. For example, if a person wanted to review a topic, he could look up the subject in his old textbook and in the appropriate chapter find a complete discussion of the topic.

- *A textbook may switch subjects.* Sometimes the topics which are to be discussed in a textbook will be disjoint. That is, the topics will not have much in common, and will have to be discussed separately. There may be no smooth way to go from one topic to the other so the textbook will just switch topics.

- *The discussions in a textbook need not be linear.* That is, there may be many intertwined ideas to be discussed which will not fit into any strict hierarchy. These ideas will have to be presented in a sort of breadth-first manner.

- *A textbook allows different viewpoints.* A textbook will often present material in several different ways to try to make subtle points more understandable. The different viewpoints give the student more insight on the problems being discussed.

- *A textbook will often contain half-truths.* These are included to shield the student from material which is too complex to be understood at that point. These white lies help the reader to gain the knowledge incrementally so that he may gradually come to understand the subject matter.

- *A textbook has historic notes.* The historic notes in a chapter of a textbook is also valuable in providing insights to the subject material. These sections often provide the background information which explains the motivations for interest in the subject. These sections also may provide clues about parts of the subject which seem awkward.

- *A textbook has a glossary and an index.* The glossary helps the reader to understand any unfamiliar terms, and the index indicates where the reader can look to find out more about a given term or concept.

### 6.1.2 Chapter and Verse

### 6.1.3 Characteristics of the textbook approach

### 6.1.3.1 Each chapter is an understandable unit

The main emphasis of the chapter and verse method of program description is in understandability. The most important aspect of understandability is that each chapter, and each section within a chapter should be an understandable unit. These units have a single idea to describe, and explain the details which are necessary to understand the idea fully. Of course, some chapters have earlier chapters as pre-requisites if background information is required.

It is desirable to have each chapter and section be a unit in itself so that one is able to read just the units of interest, and still gain a complete understanding of the material covered in the unit. That way, the description is useful to someone who only has to know about one aspect of the program. The chapter and section organization also makes clear what material is covered where, so that the reader of the description knows where to find the information needed. It is not necessary for someone to read the entire document if he only wants to know which algorithms were used to solve certain parts of the problem.

Since the material is presented in brief, understandable units it is easy to read and comprehend. The reader is not bombarded with great amounts of information in each section -- the information is spread out among sections to allow incremental understanding.

### 6.1.3.2 Non-linearity

Chapter and verse allows a comprehensive description of a program from many different viewpoints. It is possible to have more than one chapter or section which covers basically the same material but from different perspectives. The basic framework of the TFORMAT description provides a way to include all aspects of the program information, from the high level description of the problem to the implementation of the particular algorithm used, to the error conditions signalled by the program. Each different chapter and section of the description concentrates on one subject at a time and thoroughly describes and explains that subject without confusion from superfluous information.

### 6.1.3.3 White lies

"White lies" are used to convey information understandably. A white lie helps in presenting information when it is interdependent on other information. Some subset of the information must be described before other information on which it depends, so the initial description might not be the full truth. The complete truth comes when the reader has enough background to understand the whole truth. In the mean time, the white lies give the reader a picture of the situation which help him understand the material in light of what he knows. White lies are also useful in situations where the information to be passed on is so complex that the reader should be given the information in small doses. Then the reader is given an incremental understanding of the subject and is not overwhelmed. The white lies are resolved when the reader is advanced enough in his knowledge to understand the truth easily.

## 6.1.4 Limitations, problems

There are several undesirable characteristics of the chapter and verse program description. This type of description is long, and rather hard to write. There is also a problem with ordering of chapters. The order in which chapters appear does not necessarily indicate an ordering of the material contained in the

chapters, but a reader might not realize this.

### 6.1.4.1 Textbooks are hard to write

A chapter and verse program description tends to be hard to write. The reasons for this are similar to the reasons for the difficulty in writing textbooks. A lot of material is covered and explained in the description. Much of the information depends on other material which must be explained, so there is difficulty in deciding what information goes first, and how the information should be presented to best ensure its comprehensibility (and continuity where necessary). There is a requirement that the author possess some insight into the depth of information which must be covered. That is, the author is knowledgeable on the subject, but he must explain the material in such a way that someone who is not an expert can understand. A related problem is that of deciding what information should be covered. If the author has a large amount of knowledge on the subject he must be able to decide on the information which has to be conveyed in order to promote thorough understanding in someone who may not have as complete a background in the subject.

### 6.1.4.2 The description is long

Unfortunately, a textbook-style program description tends to be as long as a textbook. The following are factors which determine the length:

- The high level descriptions generate a fair amount of text since they attempt to describe all relevant issues in the problem area. If the problem area is particularly obscure or unintuitive, the high level sections can be especially long.
- The number of distinct ideas and algorithms used dictates the number of sections which describe the algorithms and their implementation. If there are many ideas then there is a lot of explanation of the ideas and their relation to the code. There will also be text to untangle the ideas from one another.
- The complexity of the user interface and the amount of potential interaction of a user with the program can affect the length of the description since they are explained in the description.
- The amount of mixing up of ideas in the code influences the length of the description since each main idea is pieced together and related to the sections of code which implement it.

It is not clear how much the description will grow with the length of the program. The length of the high level sections probably does not depend on the length of the program as much as it does on the attributes of the problem area. A complex problem area would require a lot of high level description even if the program is relatively short. On the other hand, a very long program which has a simple conceptual basis would probably not require a high level description which is proportional to the program length.

The length of the low level sections will be more dependent on the length of the program they describe since the description at that level corresponds more directly to the code. (Of course, if the program is very straightforward or repetitive, it may not require a long description.)

### 6.1.4.3 Chapter order implies an ordering of the material

One of the major benefits of using a textbook-type organization of the material is that it does not restrict the form of the description to any particular ordering. Although there is a high-to-low level progression of chapters, the content of successive chapters need not be related obviously. Some chapters depend on previous material, but others do not depend on previous material at all, and have

little relationship with the surrounding chapters. The chapter orderings should not be taken as a strict ordering of the information which they contain.

## 6.2 On-line, interactive system

Since the information is not linear, and since the information is presented on several levels to accommodate different purposes and different levels of users, the information might best be presented as an on-line interactive system. The information would be organized in a tree-like structure which represents the relationships between sections of code, and between the code and the high level concepts. That way, any user could go through the information and receive lower and lower level knowledge about a particular area (a technique which would point out the relation between the low and the high level concepts), or the user could investigate the conceptual web by staying at the same level but looking at the many viewpoints at that particular level. The interactive system would show interdependencies and would allow the user to follow the web in whatever manner best suits his needs. He would still be able to see the broad picture which would allow him to gain the understanding which this system intends to promote.

# 7. THE EXAMPLE DESCRIPTION

## 7.1 What types of chapters are needed?

The high level chapters describe the purpose and the objectives of the program. The discussions of the algorithms let the user know what to expect; the user knows if the program suits his needs. The algorithms along with some of the lower level chapters let the user know about the peculiarities of the program. This helps to give him insight into why the program behaves as it does, and helps him to use the program in the most efficient way to solve his problem.

For example, the chapter which explains the side effects of the program can be very helpful to the user of the program. He knows what to look out for while he is using the program, and can protect himself from unwanted effects, or identify potential problems. Conventional program descriptions do not include a discussion of the side effects of a program.

Each chapter must be an understandable unit, but it does not have any imposed structure other than that. Many of the typical chapters found in a chapter and verse description are recognizable as "high level chapters" or "middle level chapters" or "low level chapters".

A high level chapter explains the general concepts involved in the problem to be solved. These chapters include explanations of the terminology used in the domain of the problem. These chapters also give background information in the problem domain so that the problem may be understood. These chapters do not contain any code, or any explanation of the actual code. They are meant to give general information to a reader of any knowledge level, and do not require any knowledge of programming, or much technical knowledge. These chapters are used for several purposes. One of the purposes is to give the program developers a feel for the problem domain and the roots of the problem. Another purpose is to give an introduction to the program.

A middle level chapter is a little lower level than the high level chapters in that it begins to demonstrate some consideration of the solution of the problem. These chapters may give algorithms, etc. They are closer to the code than the very high level descriptions of the problem domain found in the high level chapters.

The low level chapters get down to the level of the code. These chapters explain how the algorithms are implemented, describe error signalling in the code, etc.

## 7.2 Categories of description

There are several categories of description which occur throughout the document. These are useful for many different chapters, high level and low level. These categories are "peculiarities", "provisions for change", "algorithms", "expected behavior", "optimizations", "paths not taken", and "viewpoints".

### 7.2.1 Peculiarities

Peculiarities are properties which are not the expected properties. In this description the peculiarities are usually unexpected results or subtleties in the program behavior. For example, in the text formatter example, it is known that when a command is encountered a break line occurs. It is not unreasonable to assume that if two commands are encountered then two break lines will occur. Since this is not the case, the actual situation is reported in the peculiarities section of the description.

The peculiarities section provides a place to point out aspects of the program's behavior which are not immediately obvious. The special section for this purpose helps to ensure that these subtle, but often important, points are not overlooked. The section turns out to be useful in low level chapters as

well as in high level chapters. In high level chapters the peculiarities are usually algorithmic oddities, or results of the program which are counter-intuitive. In lower level chapters peculiarities are usually strange happenings which occur due to some property of the particular code. Peculiarities might also include tricky algorithms if some unexpected event might result or if it may have an effect on future manipulations to the program.

### 7.2.2 Provisions for change

Provisions for change tell the reader what provisions have already been made to make modification of the program easy. This section is a benefit to the program modifier and also to the user who wants to determine the potential for customization. Provisions for change apply to the low level chapters and the high level chapters. In high level chapters the provisions for change might be features of the algorithm which allow simple changes in the operation of the algorithm. Low level chapters describe the provisions for change which were actually included in the code for the program.

### 7.2.3 Algorithms

The algorithms sections describe the high level algorithms which are used in the program. These sections do not discuss the implementation of the algorithm; their purpose is to give the reader some indication of the problem solution. The description of the algorithms helps a user to understand what problems the program is trying to solve, and how effective the solutions are in particular circumstances. For example, if the reader of the TFORMAT description is looking for a sophisticated filling algorithm he will soon realize that he should look for a different text formatter since TFORMAT uses a very simple algorithm. It is difficult for a user to come to that realization as quickly if he is studying the code or reading a description which discusses only the code and not the underlying algorithm.

### 7.2.4 Expected behavior

The expected behavior sections are of obvious benefit to the user of the program. These sections describe the results of the program which are expected in certain situations. It is also useful for the modifier of the program to understand what the expected behavior of the program is so that he can monitor the effect of his changes.

### 7.2.5 Optimizations

The TFORMAT example does not provide a good medium for demonstrating the optimizations category. The optimizations category should tell of the shortcuts which were taken, and the choices which were made in order to make the program meet requirements of speed, space, etc. This category can be extremely important, particularly in the low level chapters. In low level chapters the optimizations may be show up as obscure code, or may make the conceptual framework even less apparent than it is with code which has not been optimized. The optimizations are important to describe since these parts of the code may be a frequent target of code modifications. (If it was important to optimize these sections in the first place, it may be true that further attention to the operation of that section of code will lead to even more optimizations. In that case, it is very important for the person who is doing the modifications to know exactly how the optimizations fit into the

conceptual framework, and how the original optimizations worked.)

It was the problem of optimizations which led Wilson to the chapter and verse idea originally. Researchers require complex and lengthy computation for their data analysis. They have to have very highly optimized programs to conserve resources and allow them to analyze the data. Unfortunately, these highly optimized programs are not understandable. Even if the programs are well documented, it is only the local section of code which is documented. The optimizations are not linked to the conceptual basis of the program, nor are they well explained in their own right.

### 7.2.6 Paths not taken

A lot of insight may be gained from knowing not only how something works, but also how it does *not* work. In knowing which options *were not* used there is some light shed on the options which *were* used, and why they were used. This is why the sections on paths not taken are included in the description. After being exposed to some discussion of which algorithms were rejected and why they were rejected, the reader is less likely to misinterpret the actual operation of the algorithm since he has counter-examples with which to compare.

The reader also is made aware of the design considerations which went into the planning of the system, and may understand more clearly why the design turned out the way it did.

### 7.2.7 Viewpoints

Viewpoints provide a frame of reference for thinking about the various parts of the program. The program has multiple levels of conceptual meaning which cannot all be understood from the same point of view. One stream of thought may allow someone to understand much of the program, but other parts of the program do not fall in line with that particular line of thought. It makes much more sense to adjust the perspective to adequately reflect a part of the program than it does to abandon understanding the program in order to adhere to the original viewpoint. The viewpoints are supposed to aid in the understanding of the program; they certainly should not be so rigid as to be the cause of misinterpretations.

Viewpoints also are a factor in the existence of white lies. A white lie is a hidden category of explanations. It is the same type of section as a "peculiarities" section or a "provisions for change" section, but it is not explicitly labeled as being a "white lie". When a viewpoint does not explain some behavior of a program, it may avoid the issue (and leave it to another viewpoint) by giving an oversimplified explanation.

## 7.3 The TFORMAT example

The following subsections discuss chapters from the TFORMAT description in Appendix 1. The subsection titles correspond to one or more chapters from the example.

### 7.3.1 Chapter 1: Introduction

This chapter introduces the viewpoints which the description will take in looking at the program. The viewpoints are described and high level discussion of the viewpoints is presented. A general statement of the purpose and function of the program is given.

### 7.3.2 Chapter 2: Definition Of Important Concepts

High level concepts and terms are defined. This gives the reader a feel for the terminology and word usage as it occurs in the description. It also serves as background material for the information which follows. Only terms which are relevant to a high level description of the program are included in this chapter. Any lower level terms are defined later in the description so as not to confuse the reader with details which are not necessary for comprehension of the basic ideas. For example, terms such as "justified" and "filled" are defined, whereas terms such as "fill mode" are not. The latter term is more specific to the implementation than the former; it is not necessary to know what "fill mode" is to understand what a text formatter does. Some of these definitions include examples to make the definitions clearer and easier to understand. Also, since the definition chapter is towards the front of the description, the reader is less likely to misinterpret usage of the terms in later chapters.

### 7.3.3 Chapter 3: Justifying, Filling, Pagination, and Adjustment

This chapter provides a high level description of the algorithms which are used in the program and the peculiarities of those algorithms. The chapter also tells of some alternative algorithms which were not chosen for the program. The discussion of paths not taken gives the reader insight into what the program does not do and makes clearer what the program does.

The algorithms described here are conceptual and follow the very basic ideas of the program being described. They can be understood by non-technical as well as technical readers. The algorithms given in this chapter do not necessarily follow the structure of the program. In fact, unless some consideration was given to design the program to follow conceptual lines of thought, the algorithms will probably not always be easily spotted in the code. This is because the program code is organized in ways which are convenient for the implementation of the program, not necessarily according to the basic conceptual lines of the program.

It is important to give these algorithms so that the ideas behind the program are clear. The program itself does not indicate that there are separate fill and justification algorithms at work. Understanding what the program is doing requires high level knowledge beyond what is found in the code or the code documentation. An expert in the problem area describes what is going on -- then anyone else can gain the same insight by reading the description.

The peculiarities listed point out oddities of the algorithms which might otherwise go unnoticed. These peculiarities also illuminate aspects of the algorithms which might cause problems, thus preventing misunderstandings by the reader.

### 7.3.4 Chapter 4: In The World Of Text Formatters

This chapter relates the program to other similar programs. The strong influences of the text formatter "R" on TFORMAT are described. Knowing the roots of the TFORMAT program is helpful in understanding TFORMAT. For example, some peculiarities of TFORMAT are also peculiarities of R and were inherited from R.

### 7.3.5 Chapter 6: The Formatted Document

This chapter gives a detailed idea of what the output of the program is like. The examples of the program output let the reader see exactly what is expected from the program, and clear up possible misunderstandings about what the program produces. Included in this chapter are warnings about peculiar ways the program produces output in particular situations.

### 7.3.6 Chapter 7: Side Effects Apparent To The User

This chapter alerts users to possible side effects of the program. The chapter only includes high level side effects -- not internal side effects which go on in the code. If it is explicit how the program can affect the outside world, then it is possible for a modifier of the program to know if there will be important consequences of the modification which will effect the outside world.

This chapter helps to define the interfaces between the program and the user of the program. In the text formatter example the side effects apparent to the user are minimal, but in other programs it may become more important to users to understand how the program can change its environment.

### 7.3.7 Chapter Formatting a Document

This chapter gives a feel for the type of input which the program takes. This chapter is not a substitute for a user's manual, rather it instills some intuition about how the formatter will operate according to what it is given.

### 7.3.8 Errors

Most programs include provisions for handling error conditions. These parts of the program and the behavior which they cause are separate from the main concepts of the program. They will most likely not be included in the description of the main algorithms since the error conditions serve more as a utility to the text formatter program than as an implementation of a high level idea. Nonetheless, understanding the error handlers is important in understanding the operation of the program. Therefore, there is a chapter of the program description devoted to the explanation of the error handlers. The chapter contains information which is rather high level in addition to the code level descriptions so that the user of the program has something to consult to gain understanding about the error handlers which affect him.

### 7.3.9 Program Representations and Abstractions

The information in a chapter on the program representations and abstractions is similar to that which might be expected from a traditional program description. The chapter tells why the abstractions were chosen, and explains what the abstractions correspond to in the real world.

### 7.3.10 Program Structure

The chapters on the program structure contain information which might be found in a traditional description, but also gives a high level discussion of the program structure and its operation. These chapters give a feel for the data and control flow through the program while it is running.

### 7.3.11 Implementations Of Algorithms

Implementation of algorithms is an important chapter for anyone who will be modifying the code, or trying to understand the code. This chapter relates the high level, conceptual algorithms to the actual code. Then when code is to be modified it will be clearer how the modification fits into the ideas behind the code. Tracing modifications to their effects on the conceptual basis of the program is made possible. Also, it is easier to see what code has to be modified to effect a change when the change was conceived several levels of abstraction above the code.

### 7.3.12 Glossary and Index

The program description includes a complete glossary with extensive cross references. If a reader of the description is unsure about a term or concept he can look it up in the glossary. He will find a definition (several definitions, if appropriate) and an indication as to where he can find out more about the term.

# 8. THE FUTURE

The description is hard to do manually. The document is long and requires the author to be very knowledgeable on every aspect of the program, high level and low level. It would be ideal if there were a program environment which would have available the needed information. Then at least parts of the program description could be automatically generated. The high level descriptions could be done manually -- these sections are not terribly difficult to write and might contain more information than the program environment would keep. It is the linking of the high level to the code which would be the most beneficial contribution of an automatic system.

In order to perform the automatic description generation, the environment would have to contain some representation of the high level ideas and how they relate to the code. That is, if an accumulator is generated as part of a program, the environment has to contain some way of indicating how that accumulator fits into the conceptual framework of the program as a whole. It is not enough for the system to know that it has an accumulator -- it must also know how the accumulator is helping to implement a high level algorithm.

An automatic programming system might even be able to supply the information about paths not taken. The system might have several options from which it chose the actual implementation. It could describe the other options and tell why those options were discounted.

The programming system would also be able to recognize error handlers. It could fill in the details of the description about the code which serves the purpose of detecting and recovering error conditions.

It is not reasonable to expect a system to generate some parts of the description, like the historical notes sections. These sections require a great deal of "real-world" knowledge which the system would not have. If, however, the system had a large collection of data about the problem area it might be able to give some of the parts which require a lot of external knowledge. Even so, it would probably be more efficient to complete those sections manually.

If the program description could be automatically generated then it could also be easily updated when the code was modified. It would be the system-generated parts of the description which would be affected by changes in the code -- the parts produced manually would probably not be changed since they describe the problem area in general.

It would be a great benefit if the system could not only update the description after a modification, but also indicate the possible implications of the modifications. Then the modifier could be sure that his intentions were faithfully carried out in his modification.

It is the goal of many to be able to generate a program from a high level description. The high level information contained in this document would be one way to present the problem to an automatic programming system. The description provides the system with clues about which algorithms to look for and which to avoid. The system would provide many of the details needed for perspective in order to prevent misinterpretations of the purpose of the program.

# APPENDIX 1

CONTENTS

# PART I

## THE TEXT FORMATTER

## HIGH-LEVEL DESCRIPTION

# 1. INTRODUCTION

This document describes the program TFORMAT which is a text formatter. A text formatter arranges text so as to improve the appearance of the text. The user of the formatter supplies the text which is to be arranged, and instructions for arranging it.

A text formatting program may be most clearly understood by thinking about it from several different points of view. The reason for this is that different parts of the program act in ways which are not always consistent with just one viewpoint.

Viewpoint: A Document Changes Form

The idea behind this viewpoint is that a document is accepted by the text formatter, and then is output by the text formatter after some changes in appearance. That is, the output document is a "prettied-up" version of the input document. The document itself is sufficient to submit to the text formatter, though for extra capability there are commands available. With these commands the user can indicate to the program how to adjust certain sections of the document.

This viewpoint is useful in thinking about the behavior of the text formatter on the text of the input document.

Viewpoint: A Document is Created From Instructions

In this viewpoint, the final formatted document is created from a series of instructions provided by the user of the text formatter. The instructions include information which tells what the text of the document should be, as well as how that text should appear in the formatted document.

This viewpoint is useful for thinking about many of the text formatter functions and commands.

# 2. DEFINITIONS OF IMPORTANT CONCEPTS

DEFINITION: normal text spacing

English text which is normally spaced has one space between words, and two spaces between sentences. Newline characters separate lines. Paragraphs are separated with multiple newlines and/or spaces at the beginning of a line.

DEFINITION: line break

A line break separates words which are on one line of the output document from words which are on another line. The word immediately following a line break begins a new line.

DEFINITION: token

A token is a piece of text which is looked on as a unit. A token may be a space, a tab, a newline, or a series of characters (a word) which does not contain a space, a tab, or a newline. For example, the following symbols within the double quotation marks are tokens:

"xxxxx" "xxx.xx" ".xx" " "

DEFINITION: header

A document header is one or more lines at the top of the pages of a document. The header may be blank lines, text, or a combination of blank lines and lines with text.

DEFINITION: filled

A document is filled when normally spaced lines are approximately the same length. When the document is filled the spacing between words and characters is aesthetically pleasing.

DEFINITION: justified

A document is justified if the lines of text appear in block form. That is, all lines of text (with the possible exception of the first line in a paragraph of text) begin in the same column, and all lines of text (with the possible exception of the last line in a paragraph of text) end in the same column.

DEFINITION: paginated

A document is paginated when the document has been broken up into pages. The document is divided so that each page will have as pleasing an appearance as possible. (That is, "widow lines" are avoided, and the pages are approximately the same length.)

DEFINITION: adjusted

The combination of filling and justifying text.

# 3. JUSTIFYING, FILLING, ADJUSTMENT, AND PAGINATION

This chapter explains how the text formatter input is manipulated to produce the output document. Several of the sections discuss operations which act on a region of text. In these cases the text which is being formatted is viewed as consisting of several regions. The regions are sections of text which are visibly recognizable as belonging in a group (such as a paragraph).

## 3.1 Filling a region

Algorithm:

In the case of this text formatter, a line is filled when it has as many words as is possible without exceeding a maximum length specified for the line. The line is filled by adding a word at a time from the input document to the output document until the next word to add is longer than the room left on the line. In this case, the word that is too long starts filling the next line. This new line then receives words from the input document just as with the line before. This filling continues until the entire region of text is filled. The fill algorithm works best when the line length is long in relation to the word length. Then, statistically, the lines can be made close to the same length.

Peculiarities:

The peculiarities which exist in filling a region are due to long words in the input document. The long words cause the text formatter to act in unexpected ways since the algorithm cannot be applied as normal.

- Occasionally there is an exceptional case which arises during line filling, and the line ends up being longer than the specified maximum length. For example, if a word is longer than the maximum length of the line, then the word must be a line by itself.

- If there is only one word on a line when a word that is too long for the rest of the line is encountered, the encountered word will go on the next line, and the current line will be output with only one word, which may make this line very short in relation to the other lines.

Paths not taken:

-Widow word prevention

The algorithm used in the text formatter does not check for "widow" words. (A widow word of a paragraph appears on a different line from the rest of the paragraph.) The check is avoided so that the algorithm is left simple. This also allows the implementation of the algorithm to run more quickly. If widow words were to be eliminated, the algorithm would probably have to include look-ahead or a second pass over the output document to alter the filling of the words in the paragraph so that there are no widow words.

-Look-ahead

There is a filling strategy which entails looking ahead to ensure that no lines will have too few words on them. "Too few words" might be defined as "one or two words", or as "so few words that the combined lengths of the words is less than half the desired length of the line".

-Hyphenation

Hyphenation involves splitting words between syllables. By using hyphenation, the fill algorithm would be able to get closer to the desired line length. Hyphenation requires the program to have access to a dictionary, or to have available heuristics which would indicate the proper way to hyphenate. The look-up and/or application of the heuristic would increase the run time of the text formatter, but would allow the formatter to produce more aesthetically pleasing output documents.

## 3.2 Justifying a line

Algorithm used by this text formatter:

This formatter extends lines to the proper length by adding spaces between words. The spaces are distributed as evenly as possible between the words. In order to do the distribution, the following computation is done to find out how to insert spaces between words:

The difference is found between the actual number of characters in the line and the desired number of characters for the line. That difference is distributed evenly, if possible, among the "justifiable spaces" (spaces between words, and after the last tab of the line). If the number of justifiable spaces does not evenly divide the number of extra spaces to be added, then the remaining spaces are added one at a time to justifiable spaces, starting from the beginning or the end of the line.

Some Properties:

- A justified line may contain more spaces than were included in the input document to increase the length of a line. However, spaces are not deleted from the input text to change the length of the line.
- Leading spaces on a line are not altered during justification.
- Insertion of extra spaces for justification is done starting after the last tab of the line.
- Insertion of extra spaces is done from the beginning or end of each line, alternately from line to line. The alternation is done so that the extra spaces are distributed more evenly over the document. The extra spaces are not as noticeable when they are evenly distributed.

Peculiarities:

The process of justifying a line may result in some peculiarities in the output formatted document:

- It is likely that normal text spacing will be disrupted when text is justified. When a line is justified, extra spaces are often added to the line. These extra spaces disrupt normal text spacing.
- A justified line may be too short. This situation occurs when there is only one word to be put on a line. With only one word on the line there is no place to add any extra spaces (ie., there are no "justifiable spaces").
- A justified line may also be too short if a tab occurs before or after the last word on a line. In this case there are no justifiable spaces since only spaces after the last tab in a line are justifiable.
- A justified line may be longer than a standard justified line if the line consists of just one word which is too long to fit within the allowed margins of the text.
- Trailing spaces are stripped from a line when it is justified. (In this case the line has had spaces deleted from it.)

Paths not taken:

- The lines may be stretched, with the same amount of space inserted between each character (and space) of the line. This strategy is used to help preserve normal text spacing and separation. This approach was not taken in the case of this formatter since it was assumed that the output device which would be used would have fixed spacing, so there was no available method of changing the distance between characters by a value which is not a multiple of the width of a character.
- The spaces between words of the line may be stretched, with the same amount of space inserted between each word of the line. The strategy also has a goal of getting close to normal text spacing and separation. This approach was approximated by the actual algorithm adopted but could not be used exactly because of fixed character spacing.

## 3.3 Adjustment

The output produced by the text formatter is most aesthetically pleasing when a line to be justified has first been filled. In that case, there is a minimal number of spaces to be added to the line to make it justified. The combination of filling and justification is called "adjustment", where a document is first filled, then justified.

Algorithm:

The adjustment algorithm is the composition of the filling and justifying algorithms.

Peculiarities:

The same peculiarities found in the individual filling and justification algorithms are found in the adjustment algorithm. See above sections for those peculiarities.

Paths not taken:

An alternative adjustment algorithm which better combines the fill and justification algorithms is one which checks the results of the combination for good appearance. Such an algorithm might check for patterns which result from the combination of justification and filling, such as "rivers", and correct problems in appearance.

## 3.4 Pagination

Algorithm:

The algorithm used in pagination is essentially the same algorithm used for filling. In both cases the idea is to fit text into a given amount of space.

Pagination is done by keeping track of the number of lines put on a page so far, and going to the next page when the number of lines on a page reaches a particular limit.

After each time a line is output, the formatter checks the line number of the line just output to see if it equals the maximum number of lines allowed on a page. If the maximum number of lines allowed on a page is reached, a new page is started.

Peculiarities:

- "Widow lines" (single lines of a paragraph on a separate page from the rest of the paragraph) may occur.
- A page might end up with only one line on the page.

Paths not taken in the pagination algorithm:

- A complex pagination algorithm might check for widow lines and single lines on a page. This algorithm might make two passes over the text. The first time to arrange the text roughly, and the second time to make adjustments for widow and single lines. Some pages might not have exactly the correct number of lines, but the text would be arranged more desirably.

# 4. JUSTIFICATION OF A LINE CONTAINING TABS

When a line containing one or more tabs is justified, the line is enlarged if necessary by adding extra spaces to any justifiable spaces after the last tab in the line. The spaces are added after the last tab to ensure that the text which follows a tab actually begins on a tab setting. (If extra spaces were added after a tab and before the text which follows the tab, then the text would not begin on a tab setting.) Adding spaces after the last tab also ensures that the output document text will be aligned with the tabs as it was in the input document. (This last provision is peculiar since if the input document is to remain aligned, it is probably a table or some special form of text which the user would not want to be filled and justified.)

There is a blending of the two main viewpoints in the consideration of tabs. On one hand, the document cannot really be thought of as just changing form since an individual tab makes a difference in the way the output is formed. On the other hand, TFORMAT is not just receiving and executing instructions from the input -- the treatment of tabs requires more information about the text in the immediate area of the document than the tab itself can provide. So, the input document is changing form to produce the output document, except for when the last tab of a line is encountered in the input document. When the last tab of a line is found, the tab is treated as an instruction which indicates that spaces may be enlarged only after that tab.

# 5. IN THE WORLD OF TEXT FORMATTERS

TFORMAT is not as powerful as many text formatters. It was originally written as a student exercise. This formatter does not have enough options to be helpful where a sophisticated text formatter is necessary, but it might be useful for quick formatting jobs which can be done without professional quality text formatting.

TFORMAT has been influenced by the text formatter "R". In "R", as in TFORMAT, the commands are embedded in the input text and executed as encountered. A command which affects a body of text is in effect until another is encountered. The command names of both text formatters begin with a period and have two letters following the period. The use of two letter commands rather than names which are more mnemonic is due to the influence of "R" and not to any technical restrictions imposed by the structure of the TFORMAT program.

There are other strategies used by text formatters. The text formatter "SCRIBE" has commands which take as an argument the text on which the command is supposed to operate. The input to SCRIBE is more like a series of commands than like text with embedded commands.

# 6. INVOKING THE TEXT FORMATTER

TFORMAT is invoked with a subroutine call which includes information as to where to find the input and where the output should go, and where the error messages generated by the text formatter should go. This information is given to TFORMAT by file names. The user of the formatter specifies the name of the file which contains the input and the name of the file which should receive the output of the formatter. Also specified is a file which will receive any error messages generated by the text formatter in processing the input file.

In order to call the formatter on an input file infile.xxx, with output file outfile.xxx and error file errfile.xxx, the following CLU subroutine call is used (the double-quotation marks are necessary):

```
START_FORMAT("infile.xxx","outfile.xxx","errfile.xxx")
```

# 7. THE FORMATTED DOCUMENT

The text formatter produces a paginated document which has a header (including page number) on each page. The text in the document may be adjusted (ie. filled and justified) throughout, or it may be adjusted only in certain regions of the document.

## 7.1 Expected properties

The text formatter produces an output document which is broken up into pages with 50 lines of text. Each page has a 5 line header -- the third line of the header has the word "Page ", followed by the page number, at the left margin. (The first page is page 1.)

Lines in a section of justified text begin on the left margin (ten spaces from the left side of the page), and end on the right margin. (There are 60 spaces of text between the left and right margin.)

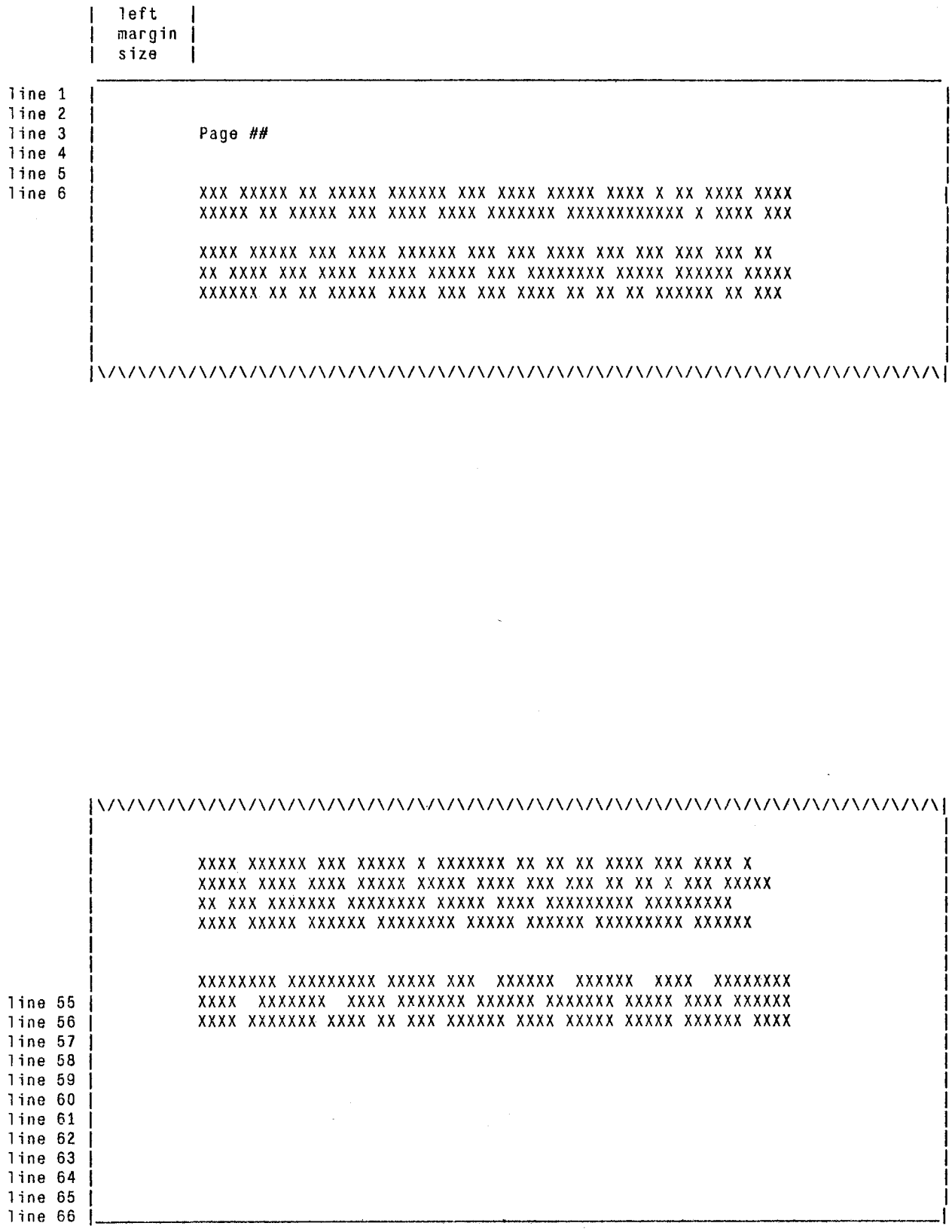
A section of text which is not justified has lines beginning on the left margin (ten spaces form the left side of the page) and continuing until a carriage return is reached. (The carriage return terminates the line.)

The following page shows an example of the form of the text formatter's output.

## 7.2 Peculiarities

This section describes output which results from input which is not normal, or which has unexpected results. Also described in this section is output which is subtle in origin.

- If there are two newlines in a row, then there will be a blank line in the output document. (This is true even when adjusting.)
- Some sections of text which are supposed to have aligned right margins might not have the proper margin alignment due to peculiarities in the fill and justify algorithms. (See the sections on peculiarities of the fill and justify algorithms in the chapter "Filling, Justification, Adjustment, and Pagination".)
- The output document usually is not produced with normal text spacing if the text is justified. (See section on justification peculiarities in the chapter "Filling, Justification, Adjustment, and Pagination".)
- The text formatter could produce a document which has a line which is on a page apart from the rest of its paragraph. (See the section on pagination peculiarities in the chapter "Filling, Justification, Adjustment, and Pagination".)


left
margin
size
line 1
line 2
line 3
line 4
line 5
line 6
Page ##
XXX XXXXX XX XXXXX XXXXXX XXX XXXX XXXXX XXXX X XX XXXX XXXX
XXXXX XX XXXXX XXX XXXX XXXX XXXXXXX XXXXXXXXXXX X XXXX XXX
XXXX XXXXX XXX XXXX XXXXXX XXX XXX XXXX XXX XXX XXX XXX XX
XX XXXX XXX XXXX XXXXX XXXXX XXX XXXXXXXX XXXXX XXXXXX XXXXX
XXXXXX XX XX XXXXX XXXX XXX XXX XXXX XX XX XX XXXXXX XX XXX
XXXX XXXXXX XXX XXXXX X XXXXXXX XX XX XX XXXX XXX XXXX X
XXXXX XXXX XXXX XXXXX XXXXX XXXX XXX XXX XX XX X XXX XXXXX
XX XXX XXXXXXX XXXXXXXX XXXXX XXXX XXXXXXXXX XXXXXXXXX
XXXX XXXXX XXXXXX XXXXXXXX XXXXX XXXXXX XXXXXXXXX XXXXXX
XXXXXXXX XXXXXXXXX XXXXX XXX XXXXXX XXXXXX XXXX XXXXXXXX
line 55
XXXX XXXXXXX XXXX XXXXXXX XXXXXX XXXXXXX XXXXX XXXX XXXXXX
line 56
XXXX XXXXXXX XXXX XX XXX XXXXXX XXXX XXXXX XXXXX XXXXXX XXXX
line 57
line 58
line 59
line 60
line 61
line 62
line 63
line 64
line 65
line 66

# 8. SIDE EFFECTS APPARENT TO THE USER

The text formatter program can not mutate (change) a file which has been given as input UNLESS

- The input file name has been given as the output file name.
- The input file name has been given as the error file name.

If the input file name is given as either the output file name or the error file name, the information which is to be sent to that file will be appended to the input file.

# PART II

# THE TEXT FORMATTER

# MIDDLE-LEVEL DESCRIPTION

# 9. POINTS OF VIEW

Viewpoint: Regions

The input to the formatter is made up of regions of text. The regions are output in the format requested for the region. This is a useful viewpoint for thinking about the algorithms which the text formatter uses -- it was introduced in the chapter FILLING, JUSTIFYING, ADJUSTMENT, AND PAGINATION. The viewpoint is helpful in that it allows the operation of the text formatter to be described abstractly. In the lower level chapters, where descriptions get more detailed, other viewpoints from which to view the program and its operation will be more useful.

Viewpoint: One token in, one token out

A token is read from the input document and put into the output document. If the token is a valid command, then the command is executed. (Command tokens are not put into the output document.)

The text formatter manipulates the tokens so that lines and pages of the output document have the appropriate number of tokens. When a page contains a specified number of newline tokens, a new page is started. The text formatter puts the header on the new page. It then resumes reading tokens from the input and uses the new page for output. This viewpoint is helpful when thinking about how the text formatter reads in the input and constructs the output document. The viewpoint is also good for detailed descriptions in lower level descriptions of the TFORMAT program. As will be seen in the chapter COMMANDS, commands may be seen from this viewpoint since commands are tokens, and only one token is needed to affect the the operation of the text formatter. The text formatter is reading tokens and deciding what to do based on the tokens encountered. The fill algorithm may also be viewed from the token point of view since the output document is filled word by word, with filling decisions based on the length of a word (which is a token).

Viewpoint: One line in, one line out

Lines are read from the input document and manipulated by the formatter to make lines for the output document. The input lines may be put directly into the output document, or may be broken up and joined with other pieces of lines to make the output document. When enough lines have been output on the current page, a new page is started. The header lines are put on the new page in the output document. The text formatter then resumes reading and outputting lines.

This viewpoint is useful in thinking about how the formatter works when operating on a region which is not to be changed by the formatter (ie., the region will not be filled or justified). In this case the text formatter appears to be transferring the input text to the output a line at a time. Also, while reading through the input the text formatter seems to be processing the input a line at a time when it reads command lines. The formatter takes each command line and executes the command.

(The text formatter actually works one line in, one line out. The input is accepted a line at a time, and the output document is output one line at a time. The output lines will not necessarily be the same as the input lines unless the lines are in a region whose format is not being changed.)

# 10. FORMATTING A DOCUMENT

The major functions of the text formatter -- formatting text with adjustment or without adjustment -- are done to regions of text. To perform these functions on a region, that region should be marked in the input with an indicator of what function is to be performed. The indicators used in this text formatter are the commands ".fi" (for adjustment) and ".nf" (for no adjustment). For example, in the following example the region of text shown will be adjusted:

```
.fi
xxxx x xxxxxx xxxx xxxxxx xxx xx xx
xxxxxxxx x xx x xxx xxx x x x xx x xxxxxx
xxxxxx x x xxxxxxx xxxxx xxxxxx xx
xxxx xx x x x xxxxxx xx xxx x x xxx xx xxxxx
```

(There are no leading spaces in the input document unless there are to be leading spaces in the output document.)

There is another command which does not follow the regions point of view. This command is the .br (break line) comamand. The ".br" command may be understood most easily from the "one token in, one token out" point of view -- when the ".br" is read on in input line, the text formatter outputs a line break.

The text portions of the document may exist in the input file in any form. Note that if the text is not to be adjusted, then it will appear in the output document as it appears in the input document.

# 11. THE COMMANDS

This chapter describes the explicit commands available for use in the text formatter. Also described are the implicit commands and their results in the final formatted document. The discussion of the implicit commands introduces a change in viewpoint from "a document changes form" to "a document is created from instructions". This is due to parts of the text taking on an active role in the formatting of the input document -- elements of the text itself have become part of the set of instructions which result in the creation of the final formatted document.

## 11.1 Command invariants

- If a command is given more than once in a row, the effect is as if the command had only appeared once.
- Commands start with a period and have two letters following the period. Ex.: .xx
- Commands occur one per line.
- Commands begin at the beginning of a line of input.
- There is nothing other than the command on a line containing a command.

## 11.2 Types of Commands

Explicit commands are unambiguous commands which produce a specific result from the text formatter. These commands are used only when a particular result is desired; the explicit commands have no meaning other than their defined meaning.

Valid explicit text formatter commands:

.br
.fi
.nf

Implicit commands are parts of the text which have an effect on the operation of the text formatter as well as having their normal meaning in the text body. The implicit commands are not as recognizable as the explicit commands since they are part of the text body and have no special format. Also, the text parts which *may* act as implicit commands do not necessarily *always* act as implicit commands. Usually it is the location in the text which causes a part of the text to act as an implicit command.

The implicit commands act differently depending whether or not they are in a region of text which is being adjusted. Some implicit text formatter commands:

<newline><space>
<newline><tab>
<newline>

When the current region is being adjusted by the formatter, the formatter is said to be in "fill mode". If the current text region is not being adjusted, then the text formatter is said to be in "no-fill mode".

## 11.3 Fill Mode

The following is a description of the text formatter's interpretation of the commands when it is operating in fill mode:

Explicit commands:

.br — .br command causes a line break in the output.

.fi — The .fi command causes a line break in the output when the formatter is already in fill mode.

.nf — The .nf command causes a line break after which the text formatter will enter no-fill mode. The text following the .nf command will be transferred directly from the input to the output document unless intermediate explicit commands are encountered.

Implicit commands:

<newline><space>
Spaces indicate the beginning of a new paragraph if they are at the beginning of a line of input. Spaces also delimit words.

<newline><tab>
A tab at the beginning of an input line indicates the beginning of a new paragraph. The last tab on a line indicates where the justification of the line should begin.

<newline>
Multiple newlines indicate the beginning of a paragraph. Single newlines have no effect.

Peculiarities:

- A line break occurs when an explicit command is encountered even if the mode does not change with the explicit command.
- If two commands which affect the current mode occur one after the other, then the result is as if only the last command had been encountered.
- If a command is given more than once in a row, the effect is as if the command had only appeared once.
- The line break occurs when a .fi command is encountered even though the current mode is "fill mode" when the .fi command is encountered.

## 11.4 No-fill Mode

The text formatter's interpretation of the commands when it is operating in no-fill mode is described below:

Explicit commands:

.br — The .br command causes a line break in the output.

.fi — The .fi command causes a line break after which the text formatter will enter fill mode (the adjustment algorithm is initiated).

.nf — The .nf command causes a line break when the formatter is in no-fill mode.

Implicit commands:

<space>
Spaces delimit words.

<tab>
Tabs have no special meaning.

<newline>
Newline characters indicate the beginning of a new line of text.

Peculiarities:

- A line break occurs when an explicit command is encountered even if the mode does not change with the explicit command.
- If two commands which affect the current mode occur one after the other, then the result is as if only the last command had been encountered.
- If a command is given more than once in a row, the effect is as if the command had only appeared once.
- A line break occurs when a .nf command is found even though the current mode is "no fill mode". However, nothing happens as a result of the line break because the current line is empty. The line is empty because the last character to go into the output document was a newline. (Commands do not go into the output.) After the newline character was encountered a new line was started but it has nothing in it when the .nf command is executed.

## 11.5 Note on command names

The command names are two letters and begin with a period due to the influence of the text formatter "R" which is the basis of this formatter.

The command ".fi" stands for "fill mode", but the command actually causes text to be filled and justified. In this case "fill mode" is not a precise description of what is going on with the text. A better command name might be ".ad" for "adjust mode", since the text is adjusted when it is justified and filled.

Likewise, the command ".nf" stands for "no fill mode". The command would make more sense if its name more accurately described its effects. For example, if the command were ".na" for "not-adjust mode", then it would be clearer that this mode does not cause text to be filled or justified.

# 12. ERRORS

There are two types of error messages which the text formatter produces. There are errors signalled by the program which go to the terminal screen of the user of the text formatter. These error messages tell of errors which the program encountered which make further processing by the text formatter program impossible.

The other type of error message produced by the text formatter goes to the error file and tells about mistakes in the input.

Whether an error is signalled by the program or an error message is sent to an error file depends on the type of error which occurs in the input or in the processing of the input.

An error is signalled by the text formatter when the following errors occur:

- The text formatter is given an input file which is not readable.
- The text formatter is given an error file which is not writable.
- The text formatter is given an output file which is not writable.

An error message is sent to the error file when the following error occurs:

- The input contains a line which begins with a period, but is not a valid command.

## 12.1 Errors signalled by the program

The following errors are signalled by the formatter when the indicated conditions occur (these errors are not sent to the error file):

INVALID INPUT FILE
: signalled when the input file is not readable

INVALID OUTPUT FILE
: signalled when the output file is not writable

INVALID ERROR FILE
: signalled when the error file is not writable

## 12.2 Errors which go to the error file

The following error messages are sent to the error file whenever the indicated error situations occur:

MISSING COMMAND
sent when a line contains only a period

'xxxxxxx' not a command
sent when an unrecognized command is encountered
after a newline

A missing command is a line which begins with a period and has nothing following the period. Missing commands cause an error message to be sent to the error file but have no result on the text being formatted. The error message sent to the error file includes the line number and a message stating that there is a missing command.

For example, if the following line is encountered by the text formatter on line 13:

.

then the error file will be sent the following message:

13: missing command

An unrecognized command is a line which begins with a period but which is not ".br", ".nf", or ".fi", or ".". Unrecognized commands are ignored except that they cause an error message to be sent to the error file. The error message sent to the error file includes the line number of the unrecognized command, the command itself, and a message stating that the attempted command is not a command.

For example, if the following line is encountered by the text formatter on line twelve:

.fill

then the error file will be sent the following message:

12: '.fill' not a command

# PART III

# THE TEXT FORMATTER

# LOW-LEVEL DESCRIPTIONS

# 13. PROGRAM REPRESENTATIONS AND ABSTRACTIONS

There are three main types of objects that the text formatter recognizes: documents, lines, and words. A document is any body of text which is made up of lines of words; lines and words are analogous to English lines and words. A document is a useful concept for representing the text formatter input as a whole. The usual application for a text formatter program is to format some text which would be described as a document.Lines and words are convenient units for thinking about and manipulating text in the input. They are the obvious divisions of a document, so it is natural to think of them as being the parts of the input which the formatter manipulates.

These three types of objects -- documents, lines, and words -- are the basis of the three main data abstractions of the text formatter program.

The data abstractions are implemented as clusters. The clusters used in the text formatter program are DOC, LINE, and WORD.

DOC is a cluster which represents a document.

LINE is a cluster which represents a line in a document. The text formatter outputs a document which is made up of lines. The line is the basic unit for justification.

WORD is a cluster which represents a word in a text line or in a document. A line is made up of WORDs.

Each cluster has operations which allow observation, mutation, and construction of objects of the type which the cluster defines.

The procedural abstractions are implemented as procedures. The procedures used in the text formatter are FORMAT, DO_LINE, DO_COMMAND and DO_TEXT_LINE.

FORMAT checks the input, output, and error streams. If the streams are usable, and the input stream is not empty, FORMAT turns control over to DO_LINE.

DO_LINE takes a preliminary look at the input. If the next input line looks like it will be a command, DO_LINE turns control over to DO_COMMAND. If the input line looks like a text line, then DO_COMMAND turns control over to DO_TEXT_LINE.

DO_COMMAND checks to be sure that the command is a valid command. If the command is valid, then DO_COMMAND initiates execution of the command.

DO_TEXT_LINE processes the text on the line to put it into the output document in the proper format.

# 14. PROGRAM STRUCTURE -- HIGH LEVEL

The TFORMAT program has several data abstractions which represent a document and the parts of a document (lines and words). The program also has some procedures which direct the text formatting which is done to documents. The procedures make sure that the commands are carried out and that the output document is produced correctly from the input document.

TFORMAT takes input text which is to be text formatted. The program designates the input as a whole as a "document". The document is composed of "lines", and the lines are made up of "words". The program formats the document by performing some task on the document, or some line or word of the document.

The procedure FORMAT checks the input to be sure it is of the correct form (ie., that it is a CLU stream).

The procedure DO_LINE decides if TFORMAT should be executing a command or processing a line of text. If a command should be executed, DO_LINE passes along the necessary information to DO_COMMAND. Likewise, if there is a line of text to process DO_LINE passes the text line to DO_TEXT_LINE.

DO_COMMAND initiates execution of a command. If the command is ".br", then DO_COMMAND calls the appropriate document operation which causes a line break in the output document. If the command is ".fi" or ".nf"),then DO_COMMAND calls the appropriate document operation which causes the text to be adjusted (or not adjusted).

DO_TEXT_LINE oversees the processing of a line of text. A line of output text is created from the input document using document, line, and word operations.

DO_TEXT_LINE uses document operations to add tokens to the output document lines. If the current region is to be adjusted, then the document operations use line operations to take care of the filling and justifying. Line operations keep track of the length of the line which is being created from tokens from the input document. When the line is long enough, line operations justify the line. The finished line is then output using line and document operations.

Each time a line is added to the output document the length of the current page is checked by a document operation. This document operation starts a new page if necessary, according to the pagination algorithm.

The DOC cluster has all the necessary operations for creating the output document. This cluster adds tokens to output documents, specifies regions which are to be adjusted, and outputs document lines.

The LINE cluster has the operations which add tokens to lines, keep track of the lengths of lines (so the lines can be filled properly), and justify lines.

The WORD cluster has the operations which take a word from the input, compute the width of a word, and output a word.

# 15. PROGRAM OPERATION OVERVIEW

The procedure FORMAT accepts the stream which has the information from the input file and checks for errors in the input, output, and error streams. FORMAT also creates an instance of a DOC which represents the document that is created from the text in the input file.

FORMAT transfers control to DO_LINE which gets a line from the input file, and decides if the line is a command line or a text line. If it is a text line, control is turned over to DO_TEXT_LINE. DO_TEXT_LINE initiates the operations necessary to format the text line. If the line is a command line, DO_COMMAND receives control and causes the proper command handler to be executed.

DO_TEXT_LINE constructs formatted text from the parts of the input line it is working on. Spaces and tabs are added to a sentence of the output by using the ADD_SPACE and ADD_WORD operations. A space is added to a line by either increasing the width of already existing spaces, or by adding an instance of a space to the array which represents a line. A tab is added to a line by adding an instance of a tab to the array which represents a line.

When a character other than a space, tab, or newline is encountered in the input it is assumed that the character is part of a word. An instance of a word is created by the SCAN operation of the WORD cluster. This new word is then added to the output document with the ADD_WORD operation of the DOC cluster. At that time the output line which is under construction is checked to see if the new word will fit. If the word fits, it is added to the line. If the new word will not fit then DOC's ADD_WORD operation terminates construction of the line being processed. This finished line is justified if necessary (by the JUSTIFY operation in the LINE cluster) and is output. The word which did not fit on the last line begins a new line.

DO_COMMAND dispatches to a command handler when a valid command is found. The command handlers are DOC operations.

# 16. IMPORTANT PROGRAM VARIABLES

The following are variables which are important to the text formatter:

chars_per_line

The number of characters per line indicates the line length for justified text. This variable is determined by the program and cannot be adjusted by the user. The number of characters per line stays constant throughout the entire operation of TFORMAT.

lines_per_page

The number of lines per page indicates the page length which must be observed in pagination of the document. This variable is determined by the program and cannot be adjusted by the user. The number of lines per page stays constant throughout the entire operation of TFORMAT.

pageno

The current page number is important in doing headers, since the page number is included in the header information. The page number is incremented by TFORMAT when a new page is started.

lineno

The current line number is important in pagination so the formatter can determine whether or not to begin a new page. The line number is incremented whenever a new line is started. (The line number is set to zero when a new page is started.)

# 17. IMPLEMENTATIONS OF ALGORITHMS

## 17.1 Implementation of the fill algorithm

The ADD_WORD operation (line 0760) in the DOC cluster (line 0560) checks to see if the current mode is fill mode (line 0770). If the current mode is fill mode, then the text should be filled. ADD_WORD checks to see if the next word of text will fit on the current line (line 0780), and if so adds the word to the line (line 0840) by calling the LINE cluster's ADD_WORD operation (line 1560). The operations LENGTH (line 1750) in the LINE cluster and WIDTH (line 2680) in the WORD cluster and the value CHARS_PER_LINE (line 0650) are used to determine if the word will fit on the line currently being processed.

The LINE cluster's ADD_WORD operation (line 1560) adds a word to the current line by adding a TOKEN (which represents a word) to the AT. The AT is an array of tokens which represents a line. This occurs in line 1570. (See line 1450 for the definition of TOKEN.)

The length of the line is increased to reflect the addition of a word. This occurs in the LINE cluster's ADD_WORD operation in line 1580.

## 17.2 Implementation of the justification algorithm

The ADD_WORD operation of the DOC cluster (line 0760) checks to see if the current mode is fill mode (line 0770). If the current mode is fill mode, then the text should be justified. The justification is done by calling (line 0790) the LINE cluster's JUSTIFY operation (line 1820) on the line which is being processed, the CHARS_PER_LINE (line 0650), and R2L. (The value R2L is kept in the representation for DOC, line 0610.)

The LINE cluster's JUSTIFY operation computes the number of extra spaces which must be added to make the line have the same number of characters as CHARS_PER_LINE. This is done on line 1890.

The number of spaces which must be added to the line is divided by the number of justifiable spaces in the line. This computation determines how many spaces should be added to each justifiable space. The division is done in the LINE cluster's ENLARGE_SPACES operation (line 2240) on line 2320. ENLARGE_SPACES is called by JUSTIFY in the LINE cluster on line 1920.

The justifiable spaces are found by determining the first justifiable space -- any spaces after the first justifiable space are justifiable. The first justifiable space is found by the LINE cluster operation FIND_FIRST_JUSTIFIABLE_SPACE (line 2040) which is called by LINE's operation JUSTIFY (line 1820). If the division done in ENLARGE_SPACES (line 2240) has a remainder, then the remaining spaces are distributed among the justifiable spaces.

The distribution of spaces is done by adding one space at a time to each justifiable space starting from the right or left end of the line and going toward the other end. The distribution of the remaining spaces and the number of spaces which must be added to every justifiable space is done in the LINE cluster's ENLARGE_SPACES operation on lines 2340 - 2450.

Spaces are "added" to a justifiable space by increasing the width of the justifiable space to include the width of the new spaces. The width of the justifiable space is actually changed on line 2410.

The value R2L tells whether the remaining spaces after the division in ENLARGE_SPACES (line 2240) should be added from the beginning or the end of the line. ENLARGE_SPACES checks the value of R2L (line 2280) and sets up a loop so that spaces will be added from the correct end of the line (line 2340).

## 17.3 Implementation of the pagination algorithm

The line number is kept in the document representation in LINENO (line 0620). When a line is output by the DOC cluster's OUTPUT_LINE operation (line 1180), the line number of the output line is checked to see if it is the last line allowed on the page (line 1340). If the line is the last on the page, then D.LINENO is set to 0 (line 1360), indicating that the next line is at the top of a page. D.PAGENO is incremented by 1 (line 1370), indicating that the next page should receive any further text.

# 18. ERROR HANDLERS AND SIGNALLERS

This chapter describes the statements in the text formatter program which exist to provide useful error messages, but which serve little or no logical purpose in error-free operation of the text formatter.

Included in the sections of this chapter are references to the actual code of the text formatter program which indicate where in the program the given lines may be found.

Error signallers and handlers in FORMAT

The following error signallers and handlers occur in the procedure FORMAT (FORMAT starts on line 0010). The signals go to the start-up procedure, which calls FORMAT, and the handlers go to the text formatter's error stream. (The error stream goes to the user-specified error file.)

Signallers:

The program signals an error if it cannot read the given input stream. (See line 0020.)
The program signals an error if it cannot write onto the given output stream. (See line 0030.)
The program signals an error if it cannot write onto the given error stream. (See line 0040.)

Handlers:

FORMAT has an error handler to catch errors which occur in the processing of the document, tell the reason for the error, and give the number of the line which contains the error. (See lines 0100 - 0110.) The reason for the error is computed by DO_COMMAND (see lines 0340 and 0350) and passed up to FORMAT through DO_LINE (see line 0200).

NOTE: Line number 0080 in the FORMAT procedure is included just for the use of the above error handler. The variable "LINE" is not used anywhere else except to provide the line number to the error handler.

Error signaller in DO_LINE:

The error signaller (line 0200) in DO_LINE is used to propagate error messages from DO_COMMAND up to the procedure which called DO_LINE.

Error message from DO_COMMAND

DO_COMMAND (starts on line 0250) checks all commands for validity. If there is an invalid command, DO-COMMAND signals an error which includes an error message which indicates the reason for the error. Line 0340 issues an error signal if there is a missing command. Line 0350 issues an error signal if some unrecognizable command was encountered.

# 19. PROVISIONS FOR CHANGE

The text formatter program has provisions for changes to the program. (For example, the formatter could be changed to allow for a different line length.) By taking advantage of some of the provisions for change, the text formatter may be made more convenient for some uses. The following paragraphs indicate the program's provisions for change. Note that the changes involve changes to the program code and are not available to the user without a change in the code. Once the code has been modified, the changes will be in effect unless the code is changed again.

The number of characters per line, the number of lines per page, and the left margin size were made equates in the DOC cluster. Therefore, these values may be easily changed by changing the numerical value in the equate.

The adjustment default may easily be changed from "fill mode" to "no-fill mode". This change is done by changing the value of FILL in the CREATE operation from true to false.

Other explicit commands may be added to the text formatter by including calls to the handlers of the commands in the DO_COMMAND procedure. These calls will be included in an ELSEIF clause in the IF statement which handles all the current explicit commands.

# 20. GLOSSARY AND CROSS-REFERENCE

adjustment: the combination of filling and justifying.
See:
Justifying, Filling, Adjustment, and Pagination
adjustment
Input
Commands
.fi
.nf
.br
historical notes on command names
Provisions for change

adjust mode: the state in which the text formatter will fill and justify text.
See:
Historical Notes
Commands
historical notes on commands names

algorithm: a procedure for solving a problem.
For the algorithms used in filling, justification, pagination, and adjustment, see:
Justifying, Filling, Adjustment, and Pagination
filling a region
justifying a line
pagination
adjustment
For the implementation of these algorithms, see:
Implementation of Algorithms

cluster: an implementation in CLU of a data abstraction.
For descriptions of clusters used in the text formatter, see:
Program Structure

document: a unit of text made up of words and lines (usually refers to a large body of text, such as the text which would be in an entire input file).
See:
What is a Text Formatter?
Formatting a Document
The Formatted Document
Input
Program Structure

delimit: to bound (for example, the letter "q" delimits

the following string of letters: "qsdlfjq").
See:
Formatting a Document
viewpoint: regions
Input
Commands
implicit commands: space

error message: an indication that an error condition has occurred.
See:
Invoking the Text Formatter
Errors
Error Messages
Error Handlers and Signallers

fill mode: Fill mode is the mode in which the text in the output document is filled and justified. That is, when the text formatter is in fill mode the text in the output document is packed with words in an aesthetically pleasing way, and the lines will all begin in the same column and have the same length.
See:
Historical Notes
historical notes on mode names
The Formatted Document
peculiarities
Commands
explicit commands
.fi
.nf
peculiarities
historical notes on command names
Implementation of Algorithms
implementation of fill algorithm
Provisions for Change

filled: A document is filled when there are the maximum possible number of words on any given line of the document. The maximum possible number of words is a number which makes the line close to a specified length, but does not leave any one line with so few words that it is not aesthetically pleasing.
See:
Definitions of Important Concepts
fill mode
no-fill mode
Justifying, Filling, Adjustment, and Pagination
filling a region

adjustment
The Formatted Document
What to Give the Text Formatter
Commands
implicit commands
.fi
.nf
historical note on command names
Implementation of Algorithms
implementation of fill algorithm

header: A document header is one or more lines at the top of the pages of a document. The header may be blank lines, text, or a combination of blank lines and lines with text.
See:
Formatting a Document
viewpoint: one token in, one token out
viewpoint: one line in, one line out
The Formatted Document
expected properties
Important Program Variables
pageno

# APPENDIX 2

```
%Read the instream, processing it and placing the output on outstream and
%writing error messages on errstream

0010    format = proc(instream, outstream,errstream: stream) signals (bad_arg(string))
0020        if ~stream$can_read(instream) then signal bad_arg("input stream")
0030         elseif ~stream$can_write(outstream) then signal bad_arg("output stream")
0040         elseif ~stream$can_write(errstream) then signal bad_arg("error stream")
0050             end
0060            d: doc := doc$create(outstream)
0070            while ~stream$empty(instream) do
0080            line: int := instream.lineno
0090            do_line(instream, d)
0100               except when error(why:string):
0110                           stream$putl(errstream,int$unparse(line) || ":\t" || why)
0120                    end
0130            end
0140            doc$terminate(d)
0150        end format
```

```
%Process an input line.  The line is processed either as a text line or as a
%command line, depending upon whether or not the first character of the line
%is a period.

0160    do_line = proc(instream: stream, d: doc) signals(error(string))
0170        c: char := stream$peekc(instream)
0180        if c = '.'
0190           then do_command(instream,d)
0200                   resignal error
0210           else do_text_line(instream,d)
0220           end
0230        end do_line
0240
```

```
%Process a command line.  This procedure reads up to the first space or tab in
%a line and processes the string read as a command.  The remainder of the line
%is read and discarded.

0250    do_command = proc(instream: stream, d: doc) signals(error(string))
0260        stream$getc(instream)  % skip the period
0270        n: string := stream$gets(instream, "\t\n")
0280           except when end_of_file: n := "" end
0290        stream$getl(instream)  % read and discard remainder of input line
0300           except when end_of_file: end
0310        if n = "br" then doc$break_line(d)
0320         elseif n = "fi" then doc$set_fill(d)
0330         elseif n = "nf" then doc$set_nofill(d)
0340         elseif n = "" then signal error("missing command")
0350         else signal error("'" || n || "' not a command")
0360         end
0370        end do_command
```

```
%Process a text line.  This procedure reads one line from instream and processes
%it as a text line.  If the first character is a word-break character, then a
%line-break is caused.  If the line is empty, then a blank line is output.
%Otherwise, the words and word-break characters in the line are processed in turn.

0380    do_text_line = proc(instream: stream, d: doc)
0390        c:char := stream$getc(instream)
0400        if c = '\n'
0410           then doc$skip_line(d)  %empty input line
0420                return
0430         elseif c = ' ' cor c = '\t'
0440           then doc$break_line(d)
0450         end
0460        while c ~= '\n' do
0470            if c = ' ' then doc$add_space(d)
0480             elseif c = '\t' then doc$add_tab(d)
0490             else w: word := word$scan(c, instream)
0500                  doc$add_word(d,w)
0510             end
0520            c := stream$getc(instream)
0530            end except when end_of_file: end
0540        doc$add_newline(d)
0550        end do_text_line
```

```
0560    doc = cluster is create, add_word, add_space, add_tab, add_newline,
0570                     break_line, skip_line, set_fill, set_nofill, terminate
0580
0590        rep =  record[line:           line,    % The current line.
0600                      fill:           bool,    % True iff in fill mode.
0610                      r2l:            bool,    % True iff justify next
                                                   %line right-to-left.
0620                      lineno:         int,     % The number of lines
                                                   %output so far on page (not including header lines).
0630                      pageno:         int,     % The number of the current output page.
0640                      outstream:      stream] % The output stream.

0650        chars_per_line = 60
0660        lines_per_page = 50
0670        left_margin_size = 10



0680        create = proc(outstream : stream) returns(cvt)
0690            return(rep${line:       line$create(),
0700                       fill:        true,
0710                       r2l:         true,
0720                       lineno:      0,
0730                       pageno:      1,
0740                       outstream:   outstream})
0750            end create

0760        add_word = proc(d:cvt, w:word)
0770            if d.fill cand ~line$empty(d.line)
0780               then if line$length(d.line) + word$width(w) > chars_per_line
0790               then line$justify(d.line,chars_per_line,d.r2l)
0800                            d.r2l := ~d.r2l
0810                            output_line(d)
0820                    end
0830               end
0840            line$add_word(d.line,w)
0850            end add_word
0860
0870        add_space = proc(d: cvt)
0880            line$add_space(d.line)
0890            end add_space


0900        add_tab = proc(d: cvt)
0910            line$add_tab(d.line)
0920            end add_tab


0930        add_newline = proc(d: cvt)
0940            if ~d.fill
0950               then output_line(d)
0960               else line$add_space(d.line)
0970               end
0980            end add_newline

0990        break_line = proc(d: cvt)
1000            if ~line$empty(d.line) then output_line(d) end
1010            end break_line

1020        skip_line = proc(d: cvt)
1030            break_line(up(d))
1040            output_line(d)  %line is empty
1050            end skip_line

1060        set_fill = proc(d: cvt)
1070            break_line(up(d))
1080            d.fill := true
1090            end set_fill
```

```
1100        set_nofill = proc(d: cvt)
1110            break_line(up(d))
1120            d.fill := false
1130            end set_nofill
1140
1150        terminate = proc(d: cvt)
1160            break_line(up(d))
1170            end terminate

1180        output_line = proc (d: rep)
1190            if d.lineno = 0
1200               then  if d.pageno > 1
1210                        then stream$putc(d.outstream, '\p') end
1220                     stream$puts(d.outstream, "\n\n")  %print header
1230                     stream$putspace(d.outstream, left_margin_size)
1240                     stream$puts(d.outstream, "Page")
1250                     stream$puts(d.outstream,int$unparse(d.pageno))
1260                     stream$puts(d.outstream, "\n\n\n")
1270               end
1280            d.lineno := d.lineno + 1
1290            if ~line$empty(d.line)
1300               then stream$putspace(d.outstream, left_margin_size)
1310                    line$output(d.line,d.outstream)
1320               end
1330            stream$putc(d.outstream,'\n')
1340            if d.lineno = lines_per_page
1350               then d.r2l := true
1360                    d.lineno := 0
1370                    d.pageno := d.pageno + 1
1380               end
1390            end output_line
1400
1410        end doc
```

```
1420    line = cluster is create, add_word, add_space, add_tab, length, empty,
1430                      justify, output
1440
1450        token = variant[space:       int,    %the int is the width of the space
1460                        tab:         int,    %the int is the width of the tab
1470                        word:        word]
1480        at = array[token]
1490        rep = record[length: int,
1500                     stuff:  at]     %the contents of the line
    %no two adjacent tokens will both be spaces

1510        max_tab_width = 8

1520        create = proc() returns(cvt)
1530            return(rep${length: 0,
1540                       stuff:  at$new()})
1550            end create

1560        add_word = proc(l: cvt, w: word)
1570            at$addh(l.stuff,token$make_word(w))
1580            l.length := l.length + word$width(w)
1590            end add_word

1600        add_space = proc(l:cvt)
1610            l.length := l.length + 1
1620            tagcase at$top(l.stuff)
1630               tag space(width: int): token$change_space(at$top(l.stuff),
1640                                                         width + 1)
1650                                      return
1660               others:
1670               end except when bounds: end % empty array
1680            at$addh(l.stuff, token$make_space(1))
1690            end add_space

1700        add_tab = proc(l:cvt)
1710            width : int := max_tab_width - (l.length // max_tab_width)
1720            l.length := l.length + width
1730            at$addh(l.stuff, token$make_tab(width))
1740            end add_tab


1750        length = proc(l:cvt)returns(int)
1760            return(l.length)
1770            end length
1780

1790        empty = proc(l:cvt) returns(bool)
1800            return(l.length = 0)
1810            end empty


1820        justify = proc(l:cvt, len:int, r2l:bool)
1830            tagcase at$top(l.stuff)
1840               tag space(width:int):at$remh(l.stuff)
1850                                    l.length := l.length - width
1860               others:
1870               end except when bounds: end %empty array
1880            if l.length >= len then return end
1890            diff : int:= len - l.length
1900            first: int := find_first_justifiable_space(l)
1910               except when none: return end
1920            enlarge_spaces(l,first,diff,r2l)
1930            end justify

1940        output = proc(l:cvt, outstream: stream)
1950            for t: token in at$elements(l.stuff) do
1960                tagcase t
1970                   tag word(w:word): word$output(w,outstream)
```

```
1980                tag space,tab(width: int): stream$putspace(outstream,width)
1990                end
2000            end
2010        l.length := 0
2020        at$trim(l.stuff,1,0)
2030        end output


2040    find_first_justifiable_space = proc(l:rep) returns(int) signals(none)
2050        a: at := l.stuff
2060        if at$empty(a) then signal none end
2070        lo: int := at$low(a)
2080        hi: int := at$high(a)
2090        i: int := hi
2100        while i > lo cand ~token$is_tab(a[i]) do
2110            i := i - 1
2120            end
2130        while i <= hi cand ~token$is_word(a[i]) do
2140            i := i + 1
2150            end
2160        while i <= hi cand ~token$is_space(a[i]) do
2170            i := i + 1
2180            end
2190        if i > hi then signal none end
2200        return(i)
2210        end find_first_justifiable_space
2220
2230
2240    enlarge_spaces = proc(l:rep, first, diff: int, r2l: bool)
2250        nspaces, last : int := count_spaces(l,first)
2260        if nspaces = 0 then return end
2270        by: int := 1
2280        if r2l
2290           then by:= -1
2300                first, last := last,first
2310           end
2320        neach: int := diff / nspaces
2330        nextra: int := diff // nspaces
2340        for i: int in int$from_to_by(first,last,by) do
2350            tagcase l.stuff[i]
2360               tag space(width: int): width := width + neach
2370                                      if nextra > 0
2380                                         then width := width + 1
2390                                              nextra := nextra - 1
2400                                         end
2410                                      token$change_space(l.stuff[i],width)
2420               others:
2430               end
2450            end
2460        l.length := l.length + diff
2470        end enlarge_spaces


2480    count_spaces = proc(l: rep, idx: int) returns(int,int)
2490        count : int := 0
2500        for i: int in int$from_to(idx, at$high(l.stuff)) do
2510            tagcase l.stuff[i]
2520               tag space: count := count + 1
2530                          idx := 1
2540               others:
2550               end
2560            end
2570        return(count,idx)
2580        end count_spaces


2590    end line
```

```
2600    word = cluster is scan, width, output

2610        rep = string


2620        scan = proc(c: char, instream: stream) returns(cvt)
2630            s: string := string$c2s(c)
2640            s := s || stream$gets(instream, " \t\n")
2650               except when end_of_file: end
2660            return(s)
2670            end scan

2680        width = proc(w:cvt) returns(int)
2690            return(string$size(w))
2700            end width


2710        output = proc(w:cvt, outstream: stream)
2720            stream$puts(outstream,w)
2730            end output


2740        end word
```